Was self-admitted technical debt removal a real removal?

Was Self-Admitted Technical Debt Removal a real Removal? An In-Depth Perspective

Fiorella Zampetti
University of Sannio, Italy
fiorella.zampetti@unisannio.it

Alexander Serebrenik
Eindhoven University of Technology, The Netherlands
a.serebrenik@tue.nl

Massimiliano Di Penta
University of Sannio, Italy
dipenta@unisannio.it

ABSTRACT

Technical Debt (TD) has been defined as "code being not quite right yet", and its presence is often self-admitted by developers through comments. The purpose of such comments is to keep track of TD and appropriately address it when possible. Building on a previous quantitative investigation by Maldonado et al. on the removal of self-admitted technical debt (SATD), in this paper we perform an in-depth quantitative and qualitative study of how SATD is addressed in five Java open source projects. On the one hand, we look at whether SATD is "accidentally" removed, and the extent to which the SATD removal is being documented. We found that that (i) between 20% and 50% of SATD comments are accidentally removed while entire classes or methods are dropped, (ii) 8% of the SATD removal is acknowledged in commit messages, and (iii) while most of the changes addressing SATD require complex source code changes, very often SATD is addressed by specific changes to method calls or conditionals. Our results can be used to better plan TD management or learn patterns for addressing certain kinds of TD and provide recommendations to developers.

CCS CONCEPTS

• **Software and its engineering → Software evolution**;

ACM Reference format:
Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2018. Was Self-Admitted Technical Debt Removal a real Removal?
An In-Depth Perspective. In *Proceedings of MSR '18: 15th International Conference on Mining Software Repositories , Gothenburg, Sweden, May 28–29, 2018 (MSR '18),* 11 pages.
DOI: 10.1145/3196398.3196423

1 INTRODUCTION

During software development activities it frequently happens that developers push code that is not in right shape yet. This can occur for several reasons, including pressure to release new features, need for quickly patching faulty code or lack of suitable components needed to implement certain features. The presence of "not quite right code which we postpone making it right" has been referred as Technical Debt (TD) by Cunningham [10].

MSR '18, Gothenburg, Sweden

DOI: 10.1145/3196398.3196423

The TD awareness is a key point to manage it [13]. Luckily, Potdar and Shihab [27] have observed that often developers tend to "self-admit" TD. This is done by inserting a comment near the source code being affected by TD, for example indicating a "TODO" and/or "FIXME", that the current code is actually a "hack" or, more generally, annotating that the code needs some improvement or refactoring at least. In order to help keep tracking of SATD, various authors have proposed approaches to automatically identify SATD comments, using regular expressions [5] or Natural Language Processing (NLP) [21]. Also, recently there have been attempts to identify, based on previous SATD and on source code features, where there could be "TD that should be admitted" [36]. In summary, while developers keep track of TD, and there are ways of detecting where TD has or should have been admitted, one may wonder whether anybody takes care of improving that sub-optimal code and therefore addressing the SATD.

Bavota and Russo [5] and Maldonado *et al.* [11] have studied the extent to which SATDs are removed. In both studies removal has been interpreted as removal of the comments reflecting SATD rather than removal of the code affected by the SATD. Maldonado *et al.* [11] have also surveyed developers involved in the introduction and/or removal of technical debt finding that SATD is predominantly removed when bugs are fixed or new features are added.

While previous work has quantitatively studied the extent to which SATD comments disappear, to the best of our knowledge the existing literature lacks a deep and systematic analysis of *how "self-admitted" technical debt has been removed.* This is relevant for several reasons. First, it is useful for estimating the effort to allocate for the improvement of the code quality. In other words, does the improvement require massive code rewriting, refactoring operations, library replacement/adaptation or is it simply matter of making the implementation more robust by changing some preconditions?

Second, the identification of recurring SATD-fixing patterns might be useful, at least in bounded circumstances, to recommend possible solutions. For example, there are cases when the code might have been made more robust by adding a null check, simplifying a complex condition, or replacing an API with an alternative one.

Last, but not least, SATD removal can be accidental, in other words, the SATD disappears because the related code is no longer in the system. It is indeed possible that, as it was also found for code smells [32], the main reason for SATD removal is not developers intentionally taking care of it, but because the code is simply no longer there. Also, it can happen that the "admission" (*i.e.,* the SATD comment) has been dropped, while the code still remains unchanged. This either means that what it was foreseen as a problem

turned out to be a suitable solution, or that developers decided to "live with that", and lose track of the TD presence.

This paper aims at performing a quantitative and qualitative in-depth analysis of whether and how SATD has been addressed in five Java open source projects. We build on top of findings of Maldonado *et al.* [11] and start from their SATD removal dataset.

First, by analyzing the changes occurred when the SATD comment disappeared (or was changed), we identify whether the it was removed "by accident" because the code was removed, moved somewhere else, and whether the SATD comment was modified.

Second, we investigate the extent to which the SATD removal is documented in commit messages, and how easy would it be to keep track of such a cross-reference. In doing so, and to triangulate previous findings of Maldonado *et al.* [11] (achieved through a survey, without analyzing source code changes) we also analyze the extent to which SATD is removed in the context of bug fixing.

Third, we analyze what kinds of changes occurred in the source code when the SATD was addressed. To this aim, we first perform a fine-grained analysis of changed source code elements using GumTree [14]. Then, we focus on particularly relevant code changes, *i.e.,* API replacement and changes to control-flow statements. Indeed, changes affecting control flow and APIs are among common change types identified by Fluri *et al.* [16]. More specifically, we isolate these kinds of changes and categorize them—using a card-sorting approach [30]—in terms of possible reasons.

Results of the study indicate that the accidental removal of SATD comments occurs quite frequently, in up to 50% of the cases, and a fairly limited percentage (8%) of SATD removals are actually documented in commit messages. When looking at how SATD is removed, we found that in most of the cases this happens through complex changes, although recurring SATD-addressing patterns are changes to method calls or conditionals. Our results can be used to better plan TD management or learn patterns for addressing certain kinds of TD and provide recommendations to developers.

Study Dataset. The study dataset is available online[1].

2 STUDY DEFINITION AND PLANNING

The *goal* of our study is to perform a deeper analysis on the removal of SATD comments with the *purpose* of investigating its circumstances and reasons. More specifically, the study aims at addressing three research questions (RQs).

The first research question is motivated by a survey conducted by Maldonado *et al.* [11], which has shown that SATD is predominantly removed when bugs are being fixed or new features are being added, *i.e.,* not as part of the intentional effort on SATD removal. We aim at verifying this observation by means of data analysis:

RQ$_1$: *To what extent is SATD removed accidentally?*

Filtering out SATD comments related to the whole class/file containing them, we study what happens to the SATD method when the comment is removed. We consider four scenarios: (i) the whole class containing the SATD comment has been removed, (ii) the method attached to the SATD comment has been removed, (iii) the method attached to the SATD comment is unchanged and, finally, (iv) the method attached to the SATD comment has been changed.

[1] http://home.ing.unisannio.it/fiorella.zampetti/datasets/ReplicationSATDRemoval.zip

Table 1: Characteristics of the studied projects [11].

Project	# Java files	SLOC	# File versions	Contributors
Camel	15,091	800,488	254,920	289
Gerrit	3,059	222,476	53,298	270
Hadoop	8,466	996,877	79,232	160
Log4j	1,112	30,287	12,609	35
Tomcat	3,187	297,828	46,716	32

Earlier studies of SATD [5, 11] highlight that between 25% and 72% of SATD comments are removed by the same developers that have introduced them. This suggests that developers are aware of the presence of SATD and of the need to remove it. By a similar argument, we expect developers who remove SATD to be aware of the removal and reflect this in the corresponding commit message:

RQ$_2$: *Is there any documented evidence of SATD removal?*

This research question aims at analyzing commit messages for the purpose of (i) determining whether SATD is removed in the context of a bug fix (respondents to the survey of Maldonado *et al.* [11] indicated that 64% of SATD is removed when fixing bugs), and (ii) determining to what extent the commit message acknowledges the SATD removal. To this aim, on the one hand, we analyze the proportion of SATD removals due to bug fixes and, on the other hand, we analyze whether the commit message is somewhat related to what reported in the SATD comment.

Next, we perform a deeper investigation of the type of changes that have been done in the code when there is a SATD removal:

RQ$_3$: *What changes occur in the source code when developers remove SATD?*

We automatically classify the changes considering changes to (i) method signature, (ii) return statements and (iii) method body. After that, using a card-sorting procedure, we manually classify the changes looking at the addition/removal of method calls related to the inclusion/exclusion of external libraries and at the changes done on conditional statements (*e.g.,* if-else and loop).

2.1 Dataset

We rely on a curated dataset used in a recent work [11] to understand the extent to which SATD comments are removed. The dataset consists of five Java open source projects covering different application domains, sizes and number of contributors. Such projects have been chosen by Maldonado *et al.* [11] also because they were highly active and highly commented. Some characteristics of the studied projects—Files, SLOC (Source Lines of code), File versions, Contributors—are reported in Table 1. The dataset reports information about introduction and removal of SATD comments. The introduction has been identified analyzing the change history and using an NLP approach [21] while, for the removal we have traced the change history backwards identifying the commit in which the SATD comment disappeared.

Tracing SATD comments to methods. While previous work [11] studied TD removal by analyzing whether the admission (SATD comment) was removed, our focus is on source code changes occurring in such a circumstance. Indeed, since we are interested (**RQ$_3$**)

Table 2: SATD removals.

Project	# SATD	# SATD Removals	# Attached to method	% Removals
Camel	1,282	877	748	58.35
Gerrit	150	88	88	58.67
Hadoop	998	306	277	27.76
Log4j	113	96	93	82.30
Tomcat	1,184	876	777	65.63

Table 3: Results of filtering false alarms.

Project	Class Renamed	Comment Moved	Comment Changed still SATD	SATD comments
Camel	0	12	98	638
Gerrit	0	12	5	71
Hadoop	4	10	26	237
Log4j	19	4	7	63
Tomcat	0	7	88	682

in analyzing source code change patterns occurring in correspondence of SATD comments removal, we focus on method-level SATD comments ignoring those concerning a whole file/class. We consider a SATD comment to be attached to a method if the comment is contained in the method body or it immediately precedes the method definition.

First, we select from Maldonado *et al.* [11] dataset only those SATD comments for which the authors have identified a removal. As done by Maldonado *et al.*, we rely on git to track renamed or moved files [6, 17]. Finally, using the *srcML* tool [9], we locate SATD comments in the source code and trace them to methods. Table 2 reports results of the analysis described above, *i.e.,* give the total number of SATD comments in a project (from Maldonado *et al.* [11] dataset), those that were removed, and among the latter how many were attached to methods. Initial figures on the total number of SATD comments are smaller than those reported by Maldonado *et al.* [11] since we found and removed duplicates and inconsistencies.

After having identified candidate method-level SATD comments removals, we have filtered out false alarms. These are cases in which (i) there is a renaming of a file containing the comment, (ii) the comment has been moved to a different location in the code as a consequence of an Extract/Move method operation, or (iii) the comment has been syntactically changed but it still represents a SATD, *e.g.*, the comment *"//TODO improve it"* that becomes *"TODO need to be improved"* without changing the source code.

To remove false alarms, given a SATD comment c attached to a method m and removed in commit r, we compare the two versions of the file f containing m immediately before and after r. Specifically, false alarms are filtered out based on the following cases.

Class Renamed and SATD comment still present (f renamed, c not affected). We analyze the change history of f to check whether f is renamed. If f is renamed to f', through *srcML* we search in f' for a comment matching c that is attached to a method having the same signature as m.

SATD comment moved in a different method and/or class (c moved to m' or f'). We check whether, at commit r, c is no longer related to m but to m' or c is removed from f while a comment c' equal to c is added to f'. Truly, these could still have been done by chance, *e.g.,* f' exhibits the same problem as f. We also check whether, together with the comment, some source code is moved from the method $m \in f$ to $m' \in f'$ or to $m' \in f$. To this aim, we leverage GumTree [14], a fine-grained AST-based differencing analyzer. Gumtree takes as input two versions of the same Java file, obtains their abstract syntax trees (ASTs) and computes the differences between them in terms of edit actions, *i.e.,* updates, additions, deletions and moves. Specifically, whether m' has the same signature of m and 50% of the source code lines deleted in m result as added in m' we conclude that the SATD comment has been simply moved.

SATD Comment Body Changed but still SATD (c rephrased but still represents SATD). To check whether the new comment still represents a SATD, we verify whether c matches any of the 62 SATD comments' patterns provided by Potdar and Shihab [28].

Table 3 reports, for each project, the number of false alarm removals, and the number of SATD comments that we have used in order to answer our three research questions.

2.2 Data Extraction and Analysis

This subsection describes the data extraction and analysis process that we follow in order to answer our research questions.

2.2.1 To what extent is SATD removed accidentally? To address $\mathbf{RQ}_1$ we use the same procedure as adopted to create our dataset (Section 2.1). Given a commit r related to a SATD comment removal (c) and based on the differences between the two versions of the Java file f containing the method m attached to c immediately before and after r, we distinguish between the following four scenarios.

SATD comment Removal dues to Class Removal (c is removed together with f). We check in the versioning system whether the class containing the comment c has been removed in r.

SATD comment Removal dues to Method Removal (c removed together with m). We use GumTree to determine whether the method attached to c has been removed together with c. Specifically, we check whether GumTree labels the method root node as "deleted".

SATD comment Removal without changing the Method (c removed, m unchanged). Relying on GumTree we verify that no syntactic changes (*e.g.,* cosmetic changes or changes to comments are possible) occurred to method m when c has been removed (in r).

SATD comment Removal modifying the Method (c removed, m changed). We check, using GumTree, whether there is a change affecting m in r. We will deeper describe this scenario in Section 2.2.3 where we will explain how we address $\mathbf{RQ}_3$ by analyzing the type of changes made.

2.2.2 Is there any documented evidence of SATD removal? To address $\mathbf{RQ}_2$, we first determine whether SATD has been removed in the context of bug fixes. To this aim, we analyze commit messages using the approach by Fischer *et al.* [15]. Specifically, we look at commit messages in order to determine a list of regular expressions to use for classifying changes as bug-fixes. Then, in order to determine whether the commit message acknowledges the SATD removal, we (i) analyze the textual similarity between the commit

message and the comment being removed, and (ii) manually inspect cases where the commit message is similar enough to the comment.

Computing the cosine similarity between commit message and SATD comment. To compute the similarity, we first pre-process both the comment body and commit message by removing special characters (*e.g.,* punctuation), numbers, single characters and URLs. Then, we perform Snowball stemming [26] and stop-word removal (using an English stop word list)[2]. We consider as candidate true positives (*i.e.,* the commit message acknowledges the SATD removal) cases where the similarity is at least 0.3 since, looking at a random set of 300 pairs we found a high percentage of pair with a similarity lower than 0.3 that do not provide any acknowledge of the SATD comment removal.

Manual labeling. After that, we manually label all the candidate pairs, using a five-level Likert scale, ranging from "not similar at all" (1) to "very similar" (5):

(1) The SATD removal is not "documented" in the commit message, *e.g.,* the SATD comment *"// TODO: Reuse spring mail support to handle the attachment"* and the commit message *"Added unit test for mail attachments to be used for wiki documentation"*.
(2) The SATD removal is indirectly linkable to the commit message (*i.e.,* as a consequence of a different change in the same commit), *e.g.,* the SATD comment *"// Stream error //TODO reset stream"* and the commit message *"StreamError needs the stream ID (so we know which stream to close)"*.
(3) The commit message is related to the SATD comment removed but the developer adds other types of information in it (*i.e.,* details and/or context), *e.g.,* the SATD comment *"// need to check the message header"* and the commit message *"support to send the response message according to the message's header return address in CamelTargetAdapter"*.
(4) The commit message explicitly reports the SATD comment removal but the latter is done within a set of other types of changes, *e.g.,* the SATD comment *"// TODO: Clean up"* and the commit message *"Code clean-up. Fix Eclipse warnings. Implement TODOs"*.
(5) The commit message clearly reports the SATD comment removal as the only scope of the change, *e.g.,* the SATD comment *"// Unexpected ACK. Log it? //TODO"* and the commit message *"Implement a TODO: Log receipt of an unexpected ACK"*.

Each pair SATD comment and commit message is independently labeled by two of the authors who also discussed and resolved inconsistent classifications.

2.2.3 What changes occur in the source code when developers remove SATD?. **RQ**$_3$ aims at investigating the type of changes done in the source code when there is a SATD comment removal. We start by quantitatively analyzing the prevalence of the types of change made when the SATD comment is removed. To obtain more profound insights into the reason behind those changes we augment the quantitative investigation with a qualitative study of two special cases: changes associated with (i) addition/removal of API imports, and (ii) addition/removal of conditionals.

[2]http://search.cpan.org/ creamyg/Lingua-StopWords-0.09/lib/Lingua/StopWords.pm

Quantitative study. Starting from results of **RQ**$_1$, we consider only those cases in which the SATD comment removal occurs when the method source code is actually changed.

We classify the changes made to the method based on the change actions outputted by GumTree. Specifically, we classify changes as *Add/Remove Method Calls*, *Add/Remove Conditional* including if-related statements, loops and switches, *Add/Remove Try-Catch*, *Modify Method Signature* including changes in the exceptions thrown, *Modify Return Statement* and *Other*. Note that each removal could belong to more than one category identified above, *e.g.,* the SATD comment removal can involve both adding a conditional statement and modifying the method signature including a new parameter that needs validation. The *Other* category includes cases in which the method attached to the SATD comment has been modified too much to be described by one of the remaining categories. Specifically, we consider a change belonging to *Other* if over 50% of the method source code lines have been changed.

After having quantitatively analyzed the changes, we perform a qualitative analysis focusing on two types of changes: those to API changes and those related to conditionals changes. A previous study on bug-fixing patterns [25] found that method call changes and changes to conditional statements are the most frequent cases of changes for bug fixing. For what concerns SATD comments removal, we believe that API changes are relevant especially when trying to improve a feature by adopting a better API. Moreover, changes to conditionals are typically performed to make the code more robust or to handle a special case of a not yet handled feature.

Qualitative study of changes involving addition or removal of API imports. We consider as candidate API changes those in which there is a change in the imported packages and in the method calls. We limit our attention to third-party APIs, excluding changes to imports related to the project itself. Two authors independently analyze all the candidate cases, and classify their relevance to API changes as "Yes" or "No". The classification has been performed according to the following criteria (i) the method call being changed is related to a class belonging to one of the added/removed packages; and (ii) the API change is relevant to the SATD comment within its context. After the labeling, the authors discuss and resolve the inconsistent classifications.

Then, the two evaluators independently label the relevant ("Yes") cases using card-sorting [30]. The labeling is performed considering as starting labels possible software maintenance activities (*i.e.,* Feature Addition, Feature Change, Performance and Maintainability). Finally, the disagreements are discussed (involving a third author as a facilitator) and resolved. The discussion also has the purpose to merge/rename the identified categories if needed.

Qualitative study of changes involving addition or removal of conditionals. Each instance belonging to the *Add/Remove Conditional* category is independently analyzed by two of the authors in order to determine the relevance between the conditional change and the SATD comment. Specifically, the change is considered relevant when it is possible to determine a relation between the comment body and the change done. For example, given a SATD comment *// FIXME: Check for null?* we consider as relevant a change that adds an if statement to ensure a pre-condition is met before an object

Table 4: SATD removal when entities are removed or when they are changed.

Project	Class Removal	Method Removal	Method not Changed	Method Changed	Total
Camel	60 (9%)	105 (16%)	109 (17%)	364 (57%)	638
Gerrit	14 (20%)	9 (13%)	3 (4%)	45 (63%)	71
Hadoop	63 (27%)	39 (16%)	13 (5%)	122 (52%)	237
Log4j	29 (46%)	9 (14%)	1 (2%)	24 (38%)	63
Tomcat	334 (49%)	74 (11%)	50 (7%)	224 (33%)	682

is accessed or an operation is performed. Then, as in the previous case, the two evaluators together resolve the disagreements.

Finally, a card-sorting approach is used to classify the changes. In this case, the starting set of labels is inspired by the bug-fix patterns by Pan *et al.* [25] (*e.g.,* add pre-condition or add branch).

3 STUDY RESULTS

This section reports the results achieved in our study and aims at answering the three research questions formulated in Section 2.

3.1 To what extent is SATD removed accidentally?

Table 4 reports what happens in the source code (*i.e.,* in the method attached to the comment) when the SATD comment is removed. Overall, we have 1,691 SATD comment removals performed in a total of 977 different commits, of which 783 (≈ 79%) removed one SATD comment only and the remaining multiple SATD comments.

The second and third column report the number and the percentage of removals occurring when the whole class or method attached to the SATD comment is removed. In principle, if the whole entity is removed we might assume that the removal occurs "accidentally", *i.e.,* not in the context of a change having the purpose of addressing the SATD. In other words, it is removed since the affected entity does not exist any more. In $\mathbf{RQ}_2$ we analyze the commit messages made upon SATD removals, and determine the extent to which this occur, at least when the SATD comment removal is documented.

Tomcat and Log4J exhibit the highest percentages of removals due to the removal of the whole class (49% and 46% respectively), while the lowest percentage is for Camel (9%). Gerrit and Hadoop, instead, show intermediate values (20% and 27%). SATD comment removals occurring along with the removal of the related methods are relatively low and similar across projects, varying between 11% for Tomcat and 16% for Camel.

The fourth column of Table 4 reports the number of SATD comments removals occurring without changing the attached methods. This scenario represents cases in which a developer is removing the comment without addressing the TD inside the code, either because she realized it was a false alarm, she decided to live with it, or the system has evolved in a way that the problem does not apply any more (*e.g.,* a pre-condition now checked elsewhere). Percentages of such cases are relatively low (always below 8%), with the only exception for Camel, where it is 17%.

Finally, we report the number of removals occurring together with changes to the attached methods (fifth, green column). These are the most interesting ones to be deeper investigated since they likely refer to changes performed with the aim of addressing the SATD. While for Gerrit, Camel and Hadoop the majority of SATD comments removals (≥ 51%) belongs to this category, for Tomcat and Log4J it is lower than 40%.

$\mathbf{RQ}_1$ summary: in general there is a high (> 30% and in three cases the majority) percentage of SATD comments removals occurring along with source code changes. However, a large percentage of removals occur when either the whole class or the method is removed. The latter is predominant for Log4J and Tomcat. While such a percentage is not as high as for bad smell removal [32], it still indicates that SATD comments are just removed by chance when evolving software.

3.2 Is there any documented evidence of SATD removal?

First of all, we investigate the extent to which the removals occur as a result of (documented) bug fixes. Among the 997 commits related to the SATD comments removals we found only 46 cases (4.6%) that were linkable to bug-fixing. Although the 14 developers responding the survey of Maldonado *et al.* [11] identified both bug-fixing and the addition of new features as the most predominant reasons when removing SATD comments, the former seems to be quite uncommon in the analyzed projects.

In order to analyze the extent to which SATD comments removals are documented, we compute the cosine similarity between the SATD comments and the commit messages in which the comment disappeared. Looking at the cosine similarity distribution among types of removals identified in $\mathbf{RQ}_1$, we found that for both class and method removals the median value is equal to 0 with few outliers showing a similarity always lower than 0.5. The above observation helps us to confirm that the removals are likely to be "accidental" (or at least unrelated to the SATD), since the commit messages do not mention the SATD comment in the scope of the change. For the other two categories (*i.e.,* method unchanged and method changed) the median similarity is ≈ 0.2 with outliers exhibiting similarity greater than 0.5.

After filtering out cases where the cosine similarity is lower than 0.3, we obtain 148 SATD comments removals that have been manually investigated by two authors (*i.e.,* evaluators) using the five-level Likert scale defined in Section 2. Evaluators agreed in 74% of the cases, with a Krippendorff's α reliability coefficient [18] of 0.71, that is greater than 0.667, the acceptability threshold[3]. All cases of disagreements have been discussed and resolved.

Figure 1 reports for each type of change the number of SATD comments removals classified in terms of admission in the commit message. There is one case only in which the SATD comment removal along with class removal is not accidental (*i.e.,* perfect match with the commit message).

Looking at those removals occurring along with a method removal, we found five cases in which the commit message is completely unrelated to the SATD, and others five related in various ways. Therefore, there are still cases (five) for which the method removal does not necessarily indicate an "accidental" removal. For

[3]It is customary to require $\alpha > 0.8$. Where tentative solutions are still acceptable $\alpha \geq 0.667$ is the lowest limit conceivable [18]

example, in Camel there is a SATD comment mentioning *"TODO: The lookup methods could possibly be removed and replaced using other methods/logic"* for which the commit message upon its removal highlights that the lookup method has been removed.

Going to those removals performed while keeping the method unchanged, we identified 20 cases exhibiting a similarity greater than 0.3. For these cases, it is interesting to verify whether the developer removes the comment since the TD does not apply any more, or she has been decided to not address it. For example, in Tomcat there is a SATD comment requiring support for a specific functionality while, at the same time, there is no awareness on whether or not such a functionality could be implemented. The commit message addressing it clearly reports that, even if it is possible to implement the required functionality, its implementation will make the overall code "messy". In Gerrit there is a case with a comment asking for implementing a better sorting strategy using as ordering key something more meaningful than the object identifier. However, the commit message reports the removal of the comment *"Remove comment about sorting ... Sorting by the quite stable change id is reasonable. Its assignment is sequential ... "*. In Tomcat, instead, there is a SATD comment stating: *"TODO SERVLET3 async"*. The commit message upon its removal states *"Remove completed TODOs"* but the source code did not change. In this case, it is possible that the Servlet async support was implemented in previous changes. Indeed, looking at the change history, we found a previous commit for which the commit message states: *"... achieve the same aim for Servlet 3.0 async processing ... the syncs will ensure that read and write aren't processed in parallel ... "*.

Finally, as regards the removals occurred changing the method, we identified 116 out of 148 cases to be inspected (similarity > 0.3). Out of these, in 52 cases the removal is completely admitted in the commit message. For example, in Tomcat there is a comment highlighting the need for setting a parameter only whether it has not been set yet (*"TODO only set this if it's null"*). The developer in charge of address it reports as the only scope of the change: *"Fix a TODO: Only set JSpFactory if not already set"*. Indeed, the source code change only introduced a null check. Moreover, in 23 cases the developer admits the removal in the commit message even if the latter occurs with other tangled changes. In 18 cases, instead, the admission occurs paraphrasing the comment body. For example, in Tomcat there is a SATD comment removed in order to fix a bug. The latter is reported as a link to the issue tracker in the commit message. Only opening the issue, it could be possible to determine the admission. The remaining 14 SATD comments are removed as a consequence of other changes. For example, in Tomcat the SATD comment stating: *"TODO Reset stream: Stream error"* is removed in a change aimed to perform more general changes to the stream handling source code, while not specifically aimed at addressing the SATD (*"StreamError needs the stream ID so we know which stream to close"*). In such a case, even if the commit message does not report the reset as the main scope of the change, it is still possible to affirm that the removal is due to the changes in the stream handling logic.

Summarizing, out of 148 SATD comments having a cosine similarity value with the commit message addressing them greater than 0.3, we found 131 cases in which the commit message (directly or indirectly) admits the removal. However, if we consider these 131 cases out of all SATD comments being removed:

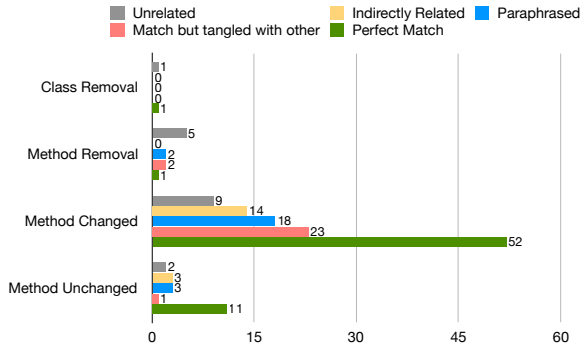

Figure 1: Categorization of admission level for different types of removals.

> **RQ$_2$ summary:** only ≈ 8% of the total set of removals are documented in commit messages. When present, the message either just mentions that the SATD is addressed, or explains why it is not the case to address it any more.

3.3 What changes occur in the source code when developers remove SATD?

Table 5 reports the types of changes applied to SATD-related methods when the SATD comment is removed. For each system we report the number and the percentage of SATD removed by applying each type of change. As explained in Section 2, the **Other** category includes the removals for which the change is too complex (*i.e.,* more than half of the method source code has been changed). Also, note that percentages do not sum up to 100 and the values for each type of change do not sum up to the total number of removals since several types of changes can occur (to the method) in the same change-set addressing the same SATD. As an example, a SATD can be addressed by adding a conditional expression and a method call.

Not surprisingly, the case having the highest percentage of occurrence is the one related to complex source code changes (**Other** category) [40%–55%]. In other words, there is a large percentage of SATD addressed by applying complex changes that involve over half of a method. Looking at the distribution of lines of code changed in the method when the removal belongs to the **Other** category, we found that on average ≈ 70% (≈ 15*LOC*) of the method body changes for addressing the SATD. The median value is 77% with an interquartile range of [66%-93%] *i.e.,* between 7 and 20 LOC.

Going to cases where specific changes occurred, most of them are related to changes in method calls [26%-45%]. For Camel the percentage is even higher than removals performed with complex changes. This result is consistent with the results of Pan *et al.* [25] for bug-fixing patterns. However, it would be interesting to investigate the reasons behind such method changes, as we will do in our qualitative analysis, especially for what concerns API changes.

The third-most frequent type of change relates to addition/removal of conditionals [11%–29%]. We have also analyzed the change history of the projects for determining the frequency of change on

Table 5: Type of changes for removing SATD-related comments.

Project	Add/Remove Method Calls	Add/Remove Conditionals	Add/Remove Try-Catch	Modify Method Signature	Modify Return	Other	Total
Camel	165 (45%)	61 (17%)	9 (3%)	36 (10%)	15 (4%)	145 (40%)	364
Gerrit	16 (36%)	8 (18%)	3 (7%)	3 (7%)	3 (7%)	23 (51%)	45
Hadoop	42 (34%)	13 (11%)	2 (2%)	6 (5%)	4 (3%)	67 (55%)	122
Log4j	8 (33%)	7 (29%)	0	0	0	10(42%)	24
Tomcat	59 (26%)	59 (26%)	5 (2%)	17 (8%)	7 (3%)	94 (42%)	224
Total	290	148	19	62	29	339	779

Table 6: Sub-classifications of Changes to Conditionals.

Project	If	Loop	Switch	Total
Camel	55	7	2	61
Gerrit	7	2	0	8
Hadoop	11	2	1	13
Log4j	7	0	1	7
Tomcat	56	5	1	59

conditionals. We found that on average 30% of the changes modify at least one conditional statement. Comparing this percentage with the one we have in the context of SATD comments removals we can conclude that developers change conditionals with a similar frequency (around 30%) in both contexts.

To provide a deeper view of changes related to conditional statements, Table 6 reports for each project the number of removals that belong to (i) `if` conditionals, (ii) loops and (iii) `switch` statements. As it can be seen from the table, the most frequent type of changes in removing SATD comments is the one aimed to modify `if` structures, while changes to loops and `switch` occurred sporadically.

Changes to *Try-Catch* instructions never occur in Log4j, while only 7% of times in Gerrit. The same happens for *return* statements. Finally, changes to the method signatures are more frequent in Camel ($\simeq$ 10%) and have never been done in Log4j.

RQ_3 quantitative summary: as also reported in previous studied aimed to identify source code patterns to fix bugs [25, 35], in removing SATD comments developers tend to apply complex changes to source code but also to modify method calls and control logic (*i.e.,* conditionals).

3.4 Qualitative analysis of SATD comments removed involving API changes

To investigate whether there is a relation between the changes done to method calls and to API imports when removing SATD comments, we focused on the 290 SATD comments removals for which there is a change in the method calls. From the above set we filtered out those removals for which there were (i) no changes in the API imports and (ii) solely changes in the API imports not related to external APIs, obtaining 174 SATD comments removals that have been manually classified by two independent evaluators. Firstly, the evaluators have determined the relevance of the API import changed with respect to the SATD comment removal. Whether the API change has been deemed relevant, they have identified a rationale behind the API import change having in mind possible software maintenance activities. We found that the evaluators agreed in 93% of the cases on whether the API imports changes

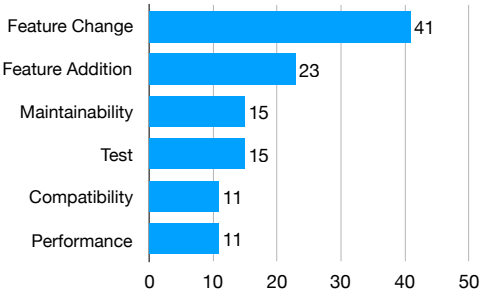

Figure 2: Categorization of SATD removals involving API changes.

were related to the SATD removal. Moreover, the Krippendorff's reliability coefficient is $\alpha = 0.92$ implying the absence of tentative solutions. In addition, we computed the reliability value in terms of kind of rationale behind the change showing that the classification is safe enough ($\alpha = 0.88$). All cases of disagreements were discussed and resolved with the third author as a facilitator when needed.

116 out of 174 changes in API imports are related to the removal of the SATD comment and the corresponding code changes. We organized the changes into six categories: **Feature Addition**, **Feature Change** (also including refactoring), **Compatibility**, **Maintainability**, **Performance** and **Test** (the SATD comment was attached to a test case).

Figure 2 summarizes the prevalence of the different categories. The largest category is **Feature Change** counting 41 SATD removals. This category includes (1) cases in which the comment suggested the addition of a piece of missed functionality whose implementation required the usage of external APIs and (2) refactoring of the existing functionality required by the SATD comment removed. As an example of the former consider the SATD comment from Tomcat requiring the validation of data received by the Servlet. In addressing it the *MessageHeader* from *javax.websocket* has been introduced in order to receive incoming messages. An example of refactoring-related SATD comment is one from Tomcat: *"TODO this needs to move to RealmBase...this is just a quick hack"* that has been addressed by removing the external APIs aimed to implement the "hack" solution. Note that we classified this refactoring as *Feature Change* rather than *Compatibility*, *Maintainability* or *Performance*, because the SATD comment does not indicate the reason why the code have to be moved to RealmBase.

Feature Addition is represented by 23 cases of SATD comments removals. For example, a SATD comment in Tomcat requires the

messages compression feature before sending them. In addressing such a SATD the developer relied on the class *Deflater* from *java.util.zip* that provides support for data compression.

In 15 cases the removal is part of a change aiming at improving the overall **Maintainability**. For example, when addressing a SATD comment asking for removing the dependency on MBeanServer, the developer also removed the no longer necessary reference to *javax.management*, *i.e.,* the package containing MBeanServer.

Similarly, we have observed 11 cases where the SATD comment is removed for improving the overall **Performance**. To illustrate this category consider a SATD comment in Gerrit highlighting the presence of a lazy behaviour in the code. Looking into the change we found that the comment was attached to a line of code instantiating an *ArrayList* object. When removing the SATD comment the developer replaced *ArrayList* with *Lists* from *com.google.common.collect*.

Finally, the remaining 11 SATD comments have been removed to ensure **Compatibility** with other technologies and/or platforms. As an example, a comment in Camel highlights the need to guarantee a proper handling of slashes in file path for the compatibility with both Unix and Windows environments. Implementing this change required the inclusion of the *java.io.File* library.

55% of SATD comments removed together with changes in external APIs belongs to **Feature Change** and **Feature Addition** categories. Therefore, SATD comments have been removed while improving/adding features.

3.5 Qualitative analysis of SATD comments removed involving "Conditional" changes

To investigate whether there is a relation between changes to conditional statements and SATD comments removals, two of the authors have manually classified each removal containing at least one change to conditional expressions. Among those 148 removals, the evaluators assigned the relevance verifying that the conditional has been modified in order to address the SATD and then, if the change in the conditional has been deemed relevant, they have classified the change starting from the bug-fixing patterns provided by Pan *et al.* [25]. The evaluators agreed in $\approx$ 92% of the cases ($\alpha = 0.794$) on whether the conditional has been changed in order to remove the TD. In terms of categories, the agreement ratio is 70% with $\alpha = 0.696$, greater than the lowest limit conceivable (0.667) [18]. After that, relying on the third author as a facilitator, we discussed and resolved all cases of disagreements.

As a result, out of 148 changes on conditionals, we have 106 relevant cases classified into 12 categories. Figure 3 reports the amount of SATD comments that have been removed for each category (*e.g.,* only in one case there is a removal of a null check in order to address the SATD).

The largest category is **Remove Code**: to address a SATD comment the developer has completely removed a code fragment containing conditional statements. This mainly happens when the SATD comment highlights the presence of a workaround, *e.g., "TODO: Remove this before release"* immediately prior to an if statement in Tomcat. Moreover, similarly to changes to external APIs,

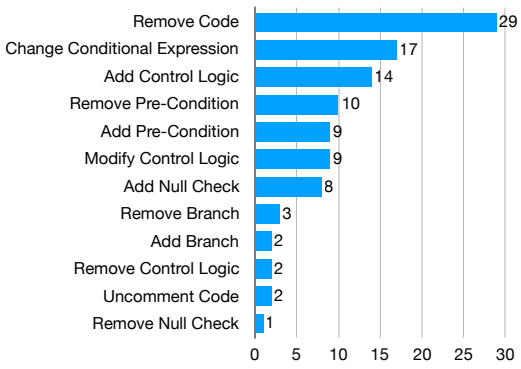

Figure 3: Categorization of SATD removals involving changes to conditional statements.

addressing a SATD might require moving a code fragment (including a conditional expression) to a different method or class, *e.g., "TODO Should be in init so we can cache"* in Gerrit.

The **Change Conditional Expression** occurs 17 times and includes cases where the conditional predicate has been changed (*e.g.,* adding a conjunct or modifying the checked parameter). For instance, in Log4j a SATD comment requires the removal of a parameter *qwIsOurs* and was located immediately before an *if*-statement checking for *(qw!=null && qwIsOurs)*. Furthermore, we also found cases in which the conditional expression is modified to improve the code maintainability: *e.g.,* in Hadoop a comment states: *"TBD not very clear"* and is inside two nested conditionals. To make the code more readable, the developer replaced the two nested conditionals with only one without modifying the functionality.

Add and **Remove Pre-condition** occur in 9 and 10 SATD comments removals, respectively. These mainly happen when there is (no longer) the need to ensure that a pre-condition is met before an operation is performed. As an example, consider the comment stating: *"... TODO: Consider storing object and only create new if changed ..."* in Camel. To address the SATD, the developer added a pre-condition checking whether the object has been changed.

The **Add/Remove Null-Check** categories can be seen as specialized cases of **Add/Remove Pre-condition**. Unsurprisingly, addition is more frequent than deletion: we have eight SATD comments addressed adding a null-check, and therefore making the code more robust, while only one SATD comment addressed removing a null-check. As an example, in Tomcat there is a comment that reports the need for setting a JSP page only if this was not previously done. We also found cases in which the null-check is required to determine whether there is the need for throwing an exception (*e.g., NullPointerException*). Moreover, the removal/addition can also occur as a consequence of refactoring. Indeed, in Log4j there is a SATD comment stating: *"TODO change when filtering refactor done"* in which the developer removes the null-check previously introduced in order to apply a patch in the code.

Add Branch and **Remove Branch** are uncommon in our dataset, counting 2 and 3 removals, respectively. Those are mainly related to the addition of a branch to cover a condition not previously considered and to the removal of a branch for freeing the code from the constraint of the conditionals. For example, there is a comment

in Camel requiring to handle non-single threaded access across connected clients, and its implementation required to add an else branch in a conditional statement. In general, and similarly to what also found for bug fixes by Pan *et al.* [25], cases of SATD addressed by adding/removing branches are uncommon, because they are related to small feature improvements that requires to handle (or to not handle any more) a special case.

Moreover, as shown in Figure 3, there are three categories related to cases in which the conditional is added/removed together with a complex block of control logic, *i.e.,* **Add/Modify/Remove Control Logic**). The latter occurs when in addressing the SATD, whole pieces of functionality are added, removed or modified. The **Remove Control Logic** occurs quite rarely in our dataset (only two instances) if compared to the addition or modification (14 and 9 respectively).

Finally, in Tomcat we have an infrequent albeit peculiar scenario we refer as **Uncomment Code**. In other words, this relates to enabling back some source code previously disabled (commented out) using an `if (false)` block.

27% of SATD comments removed together with changes in conditionals belongs to **Remove Code** category, highlighting that SATD comments are mainly used for notifying the presence of "hack" solutions or workarounds. Confirming the results of Pan *et al.* [25], the removals related to the addition/removal of branches are very uncommon.

4 THREATS TO VALIDITY

Threats to *construct validity* concern the choice of the measurements adopted in the various RQs, and the extent to which this could have affected our results. The fundamental notion SATD, defined by Potdar and Shihab as "intentional (*i.e.,* self-admitted) quick or temporary fixes (*i.e.,* technical debt)" [27] has been operationalized using source code comments. Since we reused the existing dataset of Maldonado *et al.* [11], our work inherits threats to validity of the SATD as construct by means of an NLP approach [21]. In $\mathbf{RQ}_1$ we focus on distinguishing cases where the SATD is removed by dropping the code or the entire class. While on the one hand, we discussed these cases as "accidental" removals (or in any case removals performed in the context of another change), it might be still the case that the removal was part of addressing the SATD. However, at least among the documented SATD removals ($\mathbf{RQ}_2$) we found very few of such cases (6 over 707). In $\mathbf{RQ}_2$ we used a manual analysis to evaluate whether and how a SATD comment removed is documented. Similarly, a manual analysis has been used in $\mathbf{RQ}_3$ to evaluate the purpose of the change addressing a SATD. Clearly, our assessment could be imprecise and subjective since it is based solely on what available in the commit messages and in the source code changes. We mitigated the threat by involving multiple evaluators and assessing their inter-rater agreement.

Threats to *internal validity* concern factors internal to our study that could influence our results. In particular, we have set two thresholds. In $\mathbf{RQ}_2$, we consider a threshold of 0.3 on the cosine similarity to identify candidate cases of documented removals. As explained, we calibrated this threshold by also inspecting a random sample with a lower similarity. In $\mathbf{RQ}_3$, we consider that a method was radically changed if more than half of its source code lines were changed. While this threshold may seem arbitrary, it is quite conservative because this means that we tried to classify along the other categories any change involving less than 50% of the lines, that in many cases can be anyhow substantial for a method.

Other imprecisions could be on tracing class renaming/moving ($\mathbf{RQ}_1$), where we relied on the approach used by GitHub. For the fine-grained analysis of changes, we relied on GumTree. While it might be imprecise, according to the original paper [14], it is at the moment the most accurate differencing tool and the one able to perform analyses at a finer level of granularity.

Threats to *external validity* concern the generalization of our findings. We are aware that our study is limited to a relatively small dataset of five Java projects only. However we preferred to keep the study small and (i) start from a curated dataset from a previous paper [11], and (ii) above all, deal with a number of data points allowing us to perform a thorough qualitative analysis of the observed changes. In other words, our study has the aim of being deeper than wider.

5 RELATED WORK

This section reports the literature related to (i) detection and management of TD focusing more on "self-admitted" TD; (ii) addressing quality smells over time; and (iii) identification of change patterns in software evolution and maintenance.

5.1 Detection and Management of Technical Debt and "Self-Admitted" Technical Debt

As reported by Alves *et al.* [1], technical debt (TD) can be related to different software artifacts and life-cycle activities. Previous studies made different considerations about the term "technical debt" [7, 19, 29] underlining that TD is mainly used as media between developers and managers for development issues.

Zazworka *et al.* [37] studied the impact of design TD on the quality of a software product highlighting the need for identifying and managing them closely in the development process in order to reduce their negative impact on software quality. Previous studies have investigated developers perception of TD highlighting that often their introduction is intentional [20], and, most important, the awareness is a significant problem in TD management [13].

Potdar and Shihab [28] observed that developers tend to "self-admit" technical debt (SATD) using comments highlighting the existence of somewhat temporary. Moreover, they identified 62 patterns that indicate SATD and emphasized that the presence of SATD is not uncommon in software projects. Maldonado and Shihab [22], instead, used source code comments in order to determine different types of technical debt showing that the most common type of SATD is design debt. Bavota and Russo [5] performed a qualitative analysis of SATD and created a taxonomy featuring 6 higher-level TD categories specialized in 11 sub-categories. Also, they showed that there is no correlation between SATD and code quality metrics computed at class level. They also found that (i) around 57% of SATD get removed during the change history of software projects and (ii) approximatively 63% of SATD were self-removed (*i.e.,* removed by the same developer who introduced them).

Wehaibi *et al.* [34] measured the impact of SATD on software development practices finding that the presence of self-admitted technical debt leads to a complex change in the future. From a different perspective, Zampetti *et al.* [36] developed a machine learning approach that, by leveraging on structural information (metrics or warnings raised by static analysis tools) is able to recommend developers with design TD to be admitted.

The work that is most related to ours is the one by Maldonado *et al.* [11]. They applied NLP to identify self-admitted technical debt from source code comments [21] and focused on their removal analyzing the change history of five Java open source projects. They found that (i) the majority of SATD is removed, (ii) SATD comments are mainly self-removed (54.4% on average) and (iii) the survival time varies from one project to another. Moreover, Maldonado *et al.* [11] investigated what are the activities/reasons that lead to the removal of SATD conducting a survey with 14 developers. Their results highlighted that SATD is usually removed as part of bug fixing activities (9 out of 14 respondents indicated that) and addition of new features (5 respondents).

We share with all the aforementioned papers the goal of observing how TD (and SATD in particular) is being managed and addressed. Also, we build on top of results of Maldonado *et al.* [11] as we aim at observing in detail SATD removal. Differently from their work, we first identify the amount of self-admitted technical debt that has been removed accidentally (*i.e.,* removed as a consequence of the class/method removal) and, most importantly, we aim at identifying the presence of admission of the removal in the commit message and also the presence of change-patterns in the code mainly related to their removal. Unlike what was reported in the survey of Maldonado *et al.* [11], we found a limited percentage of SATD removals occurred during bug fixing tasks ($\approx$ 5%).

Mensah *et al.* [23] clustered SATD comments for the purpose of classifying them into complex (buggy-prone) and trivial tasks, and also to estimate the amount of change required to address the SATD, which is between 10 and 25 commented LOC for complex task. While our work is not about effort estimation, our fine-grained analysis of SATD-removal changes could also be used for that purpose and therefore build better effort estimation models.

5.2 Smell removal

One kind of TD that has been studied particularly in detail by the research community is represented by code smells, perhaps also thanks to the availability of approaches and tools to identify them [2, 12, 24]. Tufano *et al.* [32] studied the evolution of code smells in over 200 open source projects. Their findings indicate that only 20% of code smells are removed, while the remaining survive in the system. Also, only 9% of the removals is due to a specific refactoring action, while the other ones are accidentals, *i.e.,* occurring just because the code is being removed. Their results are also in line with a previous study by Bavota *et al.* [4], who analyzed how refactoring activities improved source code metrics and removed smells. They found that refactoring activities does not introduce significant metric improvements and that they remove only 7% of the smells present in the source code. In a different work, Tufano *et al.* [31] conducted a similar analysis with the aim of studying the removal of test smells [33]. In this case, they found that the removal varies between 2% and 7% of the total number of smells inside the test suites.

Beyond traditional code smells, Businge *et al.* [8] have studied dependencies on internal Eclipse APIs in Eclipse third-party plug-ins. Use of those APIs is discouraged as they are solely intended for the Eclipse core and can be modified without prior notice. However, 44% of the plug-ins depend on at least one such API, and those dependencies are rarely removed. In several cases removal was accidental, *i.e.,* caused by deletion of classes depending on the API.

While our results are related to different kinds of TD, SATD in particular, which is not necessarily related to code smells, one observation still remains valid: there is a noticeable percentage (at least 40% and for two cases even the majority) of SATD that disappears just because the code is removed, and not because of intentional activities aimed at removing them.

5.3 Identification of Change Patterns

Various authors have tried to identify what are the typical change patterns occurring in the source code when developers perform certain tasks. More specifically, Pan *et al.* [25] studied change patterns occurring when fixing bugs in seven Java open source projects. They found that the most frequent changes are related to method parameter values (especially changes in method parameters) and to conditional expressions (related to adding precondition check, or adding an else branch). Our results indicate that, in terms of changes to conditionals, SATD removals follow similar proportions for some change patterns. Consistently to such a result, Yue *et al.* [35] studied repeated bug fixes, finding that (i) the phenomenon affects 15-20% of the bugs, (ii) they mostly concern cloned code, and (iii) 39% of the analyzed change patterns in those fixes are related to addition/removal of the whole `if` structures.

6 CONCLUSIONS AND FUTURE WORK

In this paper we report on the in-depth study of the removal of self-admitted technical debt. While previous studies focused on removal of comments reflecting SATD, *i.e.,* removing "admission", we deeper investigate the relation between removal of such comments and related changes on the source code. Our findings indicate that a large percentage of SATD comments removals occurs "accidentally" when either the whole class or the whole method is removed. Moreover, SATD removal is "documented" in commit messages in merely 8% of the cases. Furthermore, in addressing SATD developers tend to perform complex changes but also to modify method calls and conditionals. Finally, we designed a detailed classification of changes in external APIs and conditionals induced by the SATD comment removal.

Building on more profound insights in SATD removals and associated changes in the source code we plan to learn change-patterns for addressing certain kinds of SATD and provide recommendations to developers based on these patterns. Finally, these patterns together with the recommendations by Zampetti *et al.* [36] when TD should be self-admitted, can be built in the automatic code review bot such as Review Bot [3], not only detecting the need for TD self-admission in the code changes being reviewed but also providing suggestions for resolving the TD.

REFERENCES

[1] Nicolli SR Alves, Leilane F Ribeiro, Vivyane Caires, Thiago S Mendes, and Rodrigo O Spínola. 2014. Towards an ontology of terms on technical debt. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*. IEEE.

[2] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (2016), 1143–1191.

[3] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *International Conference on Software Engineering*. IEEE.

[4] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* (2015).

[5] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *International Conference on Mining Software Repositories*. ACM.

[6] Christian Bird, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German, and Prem Devanbu. 2009. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. IEEE.

[7] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, and others. 2010. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM.

[8] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. 2015. Eclipse API usage: the good and the bad. *Software Quality Journal* (2015).

[9] Michael L Collard, Huzefa H Kagdi, and Jonathan I Maletic. 2003. An XML-based lightweight C++ fact extractor. In *Program Comprehension, 2003. 11th IEEE International Workshop on*. IEEE.

[10] Ward Cunningham. 1992. The WyCash Portfolio Management System. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications*. ACM.

[11] Everton da S. Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. 2017. An Empirical Study on the Removal of Self-Admitted Technical Debt. In *2017 IEEE International Conference on Software Maintenance and Evolution*. IEEE.

[12] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting Code Smells using Machine Learning Techniques: Are We There Yet?. In *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE.

[13] Neil A Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L Nord, and Ian Gorton. 2015. Measure it? manage it? ignore it? software practitioners and technical debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM.

[14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM.

[15] Michael Fischer, Martin Pinzger, and Harald Gall. 2003. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE.

[16] Beat Fluri, Emanuel Giger, and Harald C. Gall. 2008. Discovering Patterns of Change Types. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*.

[17] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2011. Historage: fine-grained version control system for java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*. ACM.

[18] Klaus Krippendorff. 2012. *Content analysis: An introduction to its methodology*. Sage.

[19] Philippe Kruchten, Robert L Nord, Ipek Ozkaya, and Davide Falessi. 2013. Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. *ACM SIGSOFT Software Engineering Notes* (2013).

[20] Erin Lim, Nitin Taksande, and Carolyn Seaman. 2012. A balancing act: what software practitioners have to say about technical debt. *IEEE software* (2012).

[21] Everton Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering* (2017).

[22] Everton da S Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical debt. In *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*. IEEE.

[23] Solomon Mensah, Jacky Keung, Jeffrey Svajlenko, Kwabena Ebo Bennin, and Qing Mi. 2018. On the value of a prioritization scheme for resolving Self-admitted technical debt. *Journal of Systems and Software* (2018).

[24] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. Software Eng.* (2010).

[25] Kai Pan, Sunghun Kim, and E. James Whitehead Jr. 2009. Toward an understanding of bug fix patterns. *Empirical Software Engineering* (2009).

[26] Martin F. Porter. 2001. Snowball: A language for stemming algorithms. Published online. (October 2001).

[27] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *International Conference on Software Maintenance and Evolution*. IEEE Computer Society.

[28] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE.

[29] Carolyn Seaman and Yuepu Guo. 2011. Measuring and monitoring technical debt. *Advances in Computers* (2011).

[30] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[31] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*.

[32] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Trans. Software Eng.* (2017).

[33] Arie van Deursen, Leon Moonen, Alex Bergh, and Gerard Kok. 2001. Refactoring Test Code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*.

[34] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the impact of self-admitted technical debt on software quality. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. IEEE.

[35] Ruru Yue, Na Meng, and Qianxiang Wang. 2017. A Characterization Study of Repeated Bug Fixes. In *2017 IEEE International Conference on Software Maintenance and Evolution*.

[36] Fiorella Zampetti, Cedric Noiseux, Giuliano Antoniol, Foutse Khomh, and Massimiliano Di Penta. 2017. Recommending when Design Technical Debt Should be Self-Admitted. In *2017 IEEE International Conference on Software Maintenance and Evolution*.

[37] Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM.